第3章 初めてでも使える SystemVerilog文法ガイド

回路記述を改善し，さらに検証記述を追加

近藤 洋

ここではSystemVerilogの文法について，「記述スタイル編」と「文法ガイド編」に分けて解説する．SystemVerilogでは，テストベンチやアサーションを記述するための構文（検証記述）が用意されている．また，回路記述についても，従来より記述量が少なくなったり，ミスを誘発しにくい表現をとれるようになった．（編集部）

本稿では，SystemVerilog（バージョン3.1a）で拡張された文法について説明します．前半の「記述スタイル編」では，RTL記述，テストベンチ記述，アサーション記述の三つに分けて，記述例を用いながら解説します．予約語や文法用語については，後半の「文法ガイド編」でも解説します．合わせてご覧ください．

記述スタイル編

1 RTL記述

ここでは，RTL記述向け（つまり，論理合成可能な）SystemVerilog文法について説明します．

1.1 データ・タイプの拡張

● 基本データ・タイプ

Verilog HDLでは，ネット型（wire）とレジスタ型（reg）の2種類のデータ・タイプがありました．SystemVerilogでは，これらの基本データ・タイプに加えて，**表1**に示すデータ・タイプが追加されました．

表1　SystemVerilogで拡張された基本データ・タイプ

データ・タイプ	ビット幅	説　明
bit	1	2値（0，1）の符号なし整数．ビット幅の指定が可能
byte	8	2値（0，1）の符号付き整数（C言語のcharと同様）
shortint	16	2値（0，1）の符号付き整数（C言語のshortと同様）
int	32	2値（0，1）の符号付き整数（C言語のintと同様）
longint	64	2値（0，1）の符号付き整数（C言語のlonglongと同様）
logic	1	4値（0，1，x，z）の符号なし論理値．ビット幅の指定が可能

bit，byte，shortint，int，longintは，‘0’，‘1’の2値をとります．これらのデータ・タイプは，おもにシステム・レベル設計（RTLより上位の設計）を対象として拡張されたものです．

logicはregと同じように‘0’，‘1’，‘x’，‘z’の4値をとり，使用方法もregとほぼ同じです．ではなぜ新たに追加されたのでしょうか？ それはregの名まえから連想する回路イメージと実際の回路に差があり，混乱を招きやすかったためです．regはレジスタ型なので，フリップフロップなどの順序回路をイメージしますが，always文で記述された組み合わせ回路の出力信号もregで宣言しなければなりません．regの代わりにlogicを使用することで，このような紛らわしさがなくなりました．

もう一つ改善された点として，ネット型とレジスタ型の使用制限の緩和があります．Verilog HDLでは，always文を使って生成する信号はレジスタ型（reg）で宣言し，assign文を使って生成する信号はネット型（wire）で宣言しなければなりません．SystemVerilogではこの制限が緩和され，**表1**のデータ・タイプについてはネット型とレ

リスト1　logicを使った記述

```
logic   [1:0]   SELOUT;
logic   [3:0]   DECOUT;

assign SELOUT = SEL ? A : B;

always @( A ) begin
  case ( A )
    2'b00    : DECOUT = 4'b0001;
    2'b01    : DECOUT = 4'b0010;
    2'b10    : DECOUT = 4'b0100;
    2'b11    : DECOUT = 4'b1000;
    default  : DECOUT = 4'bxxxx;
  endcase
end
```

Verilog HDLでは，

```
wire        [1:0]SELOUT;
reg         [3:0]DECOUT;
```

リスト2　unsinged，signed，typedefの記述

```
int                      S_DATA;     // 32ビット符号付き数値
int unsigned             U_DATA;     // 32ビット符号なし数値
reg signed      [15:0]   S_REGDAT;   // 16ビット符号付き数値
reg             [15:0]   U_REGDAT;   // 16ビット符号なし数値

typedef  int unsigned       uint;
uint                A, B;            // 32ビット符号なし数値
```

リスト3　列挙型を使ったステート・マシンの記述

```
typedef enum logic [1:0] {
  NORMAL_ST, SEC_ST, MIN_ST, HOUR_ST
} state_type;

state_type CUR_STATE, NXT_STATE;

always @(posedge CLK, negedge RST_X) begin
  if (!RST_X)
    CUR_STATE <= NORMAL_ST;
  else
    CUR_STATE <= NXT_STATE;
end

always_comb begin
  case (CUR_STATE)
    NORMAL_ST :
     if (SW2)
       NXT_STATE = SEC_ST;
     else
       NXT_STATE = NORMAL_ST;
    SEC_ST:
     if (SW2)
       NXT_STATE = NORMAL_ST;
     else if (SW3)
       NXT_STATE = MIN_ST;
     else
       NXT_STATE = SEC_ST;
    MIN_ST:
     if (SW2)
       NXT_STATE = NORMAL_ST;
     else if (SW3)
       NXT_STATE = HOUR_ST;
     else
       NXT_STATE = MIN_ST;
    HOUR_ST:
     if (SW2)
       NXT_STATE = NORMAL_ST;
     else if (SW3)
       NXT_STATE = SEC_ST;
     else
       NXT_STATE = HOUR_ST;
  endcase
end
```

ジスタ型の両方で使用することができます．Verilog HDLでは信号宣言の際に，記述スタイルからregとwireの使い分けを判断していましたが，**リスト1**のようにすべてlogicで宣言することができます．

ただし，**表1**のデータ・タイプは，複数の値が同時に代入された場合に最終値を決定する解決機能がないため，複数のalways文やassign文などからの代入はできません．

● unsignedとsigned

Verilog HDL 1995では符号付き数値はintegerだけでしたが，Verilog HDL 2001では符号なし数値を符号付き数値として宣言する修飾子signedが追加されました．これにより，regなどで宣言された任意のビット数の信号に対して，符号付き数値として演算することが可能となりました．

さらにSystemVerilogでは，bitやintなどの符号付き数値を符号なし数値として宣言する修飾子unsignedが追加されています．

signedとunsignedはデータ・タイプの後ろに記述します．C言語の宣言方法と異なるので注意が必要です．

● ユーザ定義型(typedef)

SystemVerilogのデータ・タイプを用いて，ユーザが任意のデータ・タイプを定義できます．よく使用するデータ・タイプや，次に説明する列挙型などのように記述が長くなるものについては，typedefを使って定義しておいたほうが修正しやすくなり，また，記述も見やすくなります．

リスト2にunsigned，signed，およびtypedefを使った宣言の例を示します．

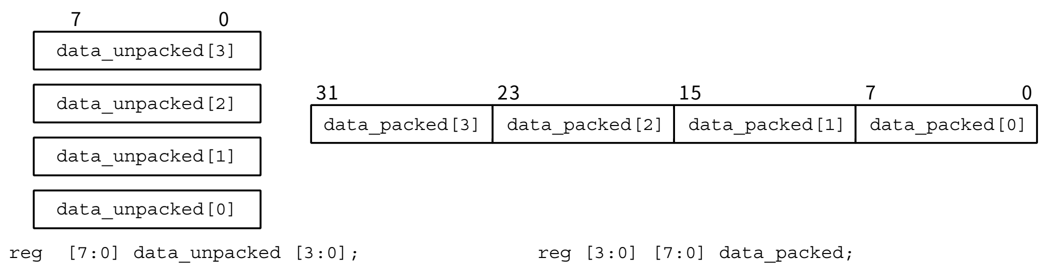
● 列挙型(enum)

列挙型は，取りうる値の集合を列挙名のリストとして宣言するデータ・タイプです．**リスト3**はステート・マシンの状態変数を列挙型で宣言した例です．列挙型を使用することで，ステート・マシンの状態名を列挙名として記述することができます．

列挙名のデータ・タイプはデフォルトでint型になりますが，**リスト3**のようにlogic[1:0]を記述することで，2ビットのlogic型にできます．

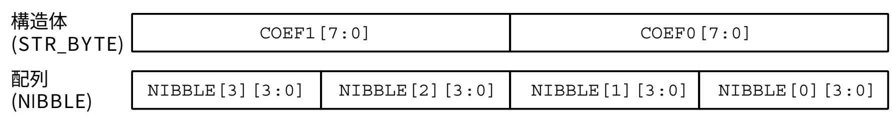
また，列挙名の値はデフォルトで指定されたデータ・タイプにより0から順番に値が割り振られますが，**リスト4**

図1
アンパック型配列とパック型配列
(a)のアンパック型配列は，8ビット幅のデータが独立して四つ存在する．(b)のパック型配列は，8ビット幅のデータが四つ連続して存在するため，32ビットのベクタとして取り扱うことができる．



(a) アンパック型配列　　(b) パック型配列

図2
レジスタ構成(FILT_DR)
FILT_DRは，8ビットのデータCOEF1，COEF0の二つのメンバからなる構造体(STR_BYTE)と，4ビット×4のパック型配列(NIBBLE)の両方のデータ・タイプでアクセス可能とする(リスト5を参照)．



リスト4
列挙名の値を指定した記述

```
typedef enum logic [3:0] {
  NORMAL_ST = 4'b0001,
  SEC_ST    = 4'b0010,
  MIN_ST    = 4'b0100,
  HOUR_ST   = 4'b1000
} state_type;
```

リスト5
パック型共用体とパック型構造体の記述

```
typedef struct packed{
  logic [7:0]  COEF1;
  logic [7:0]  COEF0;
} filtdr_type;

union packed{
  filtdr_type       STR_BYTE;
  logic [3:0][3:0] NIBBLE;
}FILT_DR;
```

のようにユーザが列挙名の後に値を指定すると，この値を変更することができます．

● アンパック型配列とパック型配列

SystemVerilogではアンパック型配列とパック型配列の2種類の配列を定義しています．従来のVerilog HDLにおける配列宣言はアンパック型配列で，新たにパック型配列が追加されました．

アンパック型配列とパック型配列は宣言のスタイルが異なります．パック型配列は，bitやlogicなどのデータ・タイプと配列名の間に配列の要素の数と各要素のビット幅を記述します．これに対してアンパック型では，要素の数を配列名の後ろに記述します．

アンパック型配列は，**図1(a)**のように個々の要素が独立して存在しますが，パック型配列は個々の要素が連続したベクタとして存在します．したがって，**図1(b)**のパック型配列には，直接32ビットのデータを代入できます．

アンパック型配列は，SystemVerilogの全データ・タイプに使用できるのに対して，パック型配列はビット単位のデータ型(logic，bit，reg，wire)，パック型配列，および次に説明するパック型構造体(struct)とパック型共用体(union)に対して宣言することができます．

● 構造体(struct)と共用体(union)

C言語ではすでになじみの深い構造体と共用体がSystem Verilogで使用できるようになりました．

構造体は，関連のあるデータを一つのグループとしてまとめるデータ・タイプです．データ・タイプやサイズが異なるデータをグループ化することができます．一方，共用体は，一つの変数に対して複数のデータ・タイプを割り当てるデータ・タイプです．共用体で宣言された変数は異なるデータ・タイプでアクセスすることができます．

構造体と共用体には，配列と同じようにアンパック型とパック型があります．デフォルトはアンパック型で，パック型にするにはstructやunionのキーワードの後にpakcedを明記します．

リスト5は，**図2**のレジスタ構成をパック型構造体とパック型共用体で記述した例です．この16ビットのレジスタFILT_DRはパック型共用体で宣言されており，共用体の中は4ビット×4のパック型配列のデータ(16ビット，

NIBBLE）とfiltdr_typeで定義された構造体（16ビット，STR_BYTE）の二つのメンバが宣言されています．したがって，FILT_DRはパック型配列（NIBBLE）でも，filtdr_typeで宣言された構造体（STR_BYTE）でもアクセスすることができます．

構造体filtdr_typeの中は，8ビットの二つの係数データCOEF0，COEF1が構造体のメンバとして宣言されています．

構造体と共用体のメンバの値の参照や代入を行う場合は，C言語と同じようにメンバ名を使用します．例えばFILT_DRのCOEF0の値を参照したい場合はFILT_DR.STR_BYTE.COEF0と記述し，FILT_DRのCOEF1の下位4ビットを参照する場合はFILT_DR.NIBBLE[2]と記述します．

構造体はアンパック型，パック型ともに論理合成可能ですが，共用体はパック型のみ論理合成可能です．また，パック型共用体は内部で宣言するデータ・タイプのビット長がすべて同じでなければなりません．

1.2　新しいalways文

Verilog HDLにおけるalways文は，組み合わせ回路と順序回路とでは記述スタイルが異なり，注意が必要でした．記述を誤ると，組み合わせ回路で記述したつもりなのに，意図しないラッチ回路を生成してしまいました．

SystemVerilogでは，設計者が意図する回路構成を明確にし，より記述をシンプルにする目的で，以下の三つのalways文が追加されました．

● always_comb

組み合わせ回路を記述するためのalways文です．always文を使って組み合わせ回路を記述する場合，以下のような記述上の注意点がありました．

- センシティビティ・リストにはすべての入力信号を記述する．
- if文，case文を使用する場合はすべての条件式を記述する．

always_combでは，まずセンシティビティ・リストの記述が不要です．また，EDAツールがalways_combで記述された回路は組み合わせ回路であることを判断してくれるため，もしif文やcase文の条件の不足によりラッチ回路となる場合でも，コンパイル時に設計者に対してメッセージが出力されます．このメッセージにより，設計者は早期に記述の誤りを見つけることができます．

もう一つの特徴は，出力の初期値を確定するため，シミュレーション開始時（時刻0）に1度だけ実行される点です．これは，入力値の初期値と出力値を保証するために行われます．always文で組み合わせ回路を記述した場合，センシティビティ・リストの信号に変化が起こった時点で初めて組み合わせ回路の出力値が確定しますが，always_combの場合は時刻0で出力の初期値が確定します．

リスト3で紹介した列挙型を用いたステート・マシンの記述では，always_combを使って次の状態を決める組み合わせ回路を記述しています．

リスト6　always_ffとalways_latchの記述

```
logic       CNT_START, CNT_END, CNTEN;
logic [2:0] CNT;
always_ff @(posedge CLK or negedge RST_X) begin
  if(!RST_X)
    CNT <= 3'b000;
  else if(CNTEN)
    CNT <= CNT + 3'b001;
end
always_latch begin
  if (!RST_X)
    CNTEN <= 1'b0;
  else if(CNT_START)
    CNTEN <= 1'b1;
  else if(CNT_END)
    CNTEN <= 1'b0;
end
```

● always_ffとalways_latch

always_ffはフリップフロップを含む順序回路を記述するためのalways文です．記述方法はalways文を使って順序回路を記述する場合とまったく同じです．

always_latchはラッチを記述するためのalways文です．always_latchもalways_combと同じように，センシティビティ・リストの記述が不要であり，シミュレーション開始時に1度だけ実行されます．

リスト6にalways_ffとalways_latchの記述例を示します．

1.3　if文とcase文の拡張

先ほどのalways文の拡張と同じように設計者の意図を明確にする目的で，if文，case文に対してuniqueとpriorityのキーワードが追加されました．ともにif文，case文の先頭に記述します．

3

リスト7 uniqueとpriorityの記述

```
logic [3:0] DIN;
logic [1:0] DOUT;

always_comb begin
  unique case ( DIN )
    4'b0001 : DOUT = 2'b00;
    4'b0010 : DOUT = 2'b01;
    4'b0100 : DOUT = 2'b10;
    4'b1000 : DOUT = 2'b11;
    default : DOUT = 2'bxx;
  endcase
end
```

(a) unique caseの記述

● 重複してはいけない
● すべての条件が記述されていなければならない

```
logic [3:0] DIN;
logic [1:0] DOUT;

always_comb begin
  unique if (DIN == 4'b0001)
    DOUT = 2'b00;
  else if (DIN == 4'b0010)
    DOUT = 2'b01;
  else if (DIN == 4'b0100)
    DOUT = 2'b10;
  else
    DOUT = 2'b11;
end
```

(b) unique ifの記述

```
logic IRQ0, IRQ1, IRQ2, IRQ3;
logic [3:0] DOUT;

always_comb begin
  priority case ( 1'b1 )
    IRQ0    : DOUT = 4'b0001;
    IRQ1    : DOUT = 4'b0010;
    IRQ2    : DOUT = 4'b0100;
    IRQ3    : DOUT = 4'b1000;
    default : DOUT = 4'bxxxx;
  endcase
end
```

(c) priority caseの記述

● すべての条件が記述されていなければならない

```
logic IRQ0, IRQ1, IRQ2, IRQ3;
logic [3:0] DOUT;

always_comb begin
  priority if (IRQ0)
    DOUT = 4'b0001;
  else if (IRQ1)
    DOUT = 4'b0010;
  else if (IRQ2)
    DOUT = 4'b0100;
  else
    DOUT = 4'b1000;
end
```

(d) priority ifの記述

● unique

uniqueは，if文やcase文で記述された回路が並列処理できることを意味します．リスト7(a)，(b)にunique caseとunique ifの記述例を示します．

uniqueのキーワードを使用する場合，以下の2点を守らなければなりません．

- 条件式(選択項目)に重複がないこと
- すべての条件が記述されていること

この二つの記述上の制約はコンパイル時やシミュレーション時にツールでチェックされ，違反した場合はメッセージが出力されます．

● priority

priorityは，if文やcase文で記述された回路が優先順位を持った回路であることを意味します．リスト7(c)，(d)にpriority caseとpriority ifの記述例を示します．

priorityのキーワードを使用する場合，以下の条件を守らなければなりません．

- すべての条件式が記述されていること

priorityが記述されている場合，uniqueと同様にツールはコンパイル時やシミュレーション時に上記のチェックを行い，違反している場合はメッセージを通知します．

1.4 モジュールの接続

Verilog HDLでは名まえによるポート接続と順番によるポート接続の2種類がありました．SystemVerilogでは新たに.nameと.*の2種類のポート接続が追加されました．

● .nameによるポート接続

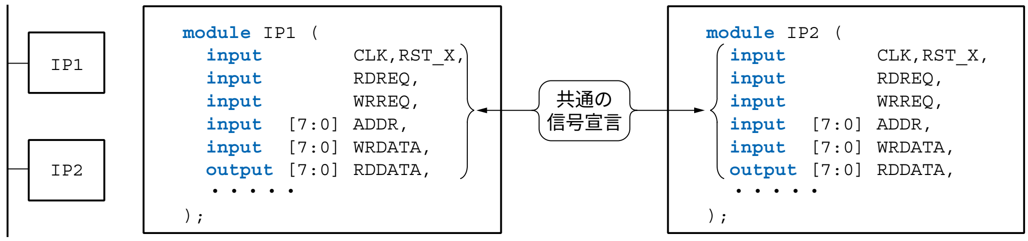
.nameは名まえによるポート接続をより簡単化したものです．モジュール接続の記述において，ポート名とネット名が同一であるケースが非常に多く見受けられます．.nameを使うと，ポート名とネット名が同一である場合はポート名だけ記述すればよく，接続するネット名を省略できます．リスト8(b)に.nameを使ったモジュール接続の例を示します．

なお，ポート名とネット名が一致しない信号については，名まえによるポート接続と組み合わせて記述することができます．

● .*によるポート接続

.*は.nameをさらに簡単にしたものです．ポート名とネット名が同一である場合，ポート名も省略して.*のみを

図3
モジュール間の共通信号
モジュール間の共通の信号は，モジュール宣言のたびにすべての信号を記述しなければならないため，記述が冗長になる．



リスト8　モジュール接続の記述

```
CNT6 CNT6 (
  .CLK     ( CLOCK  ),
  .RST_X   ( RST_X  ),
  .CNT_EN  ( CNT_EN ),
  .CNTOUT  ( CNTOUT )
);
```

(a) 名まえによるポート接続

```
CNT6 CNT6 (
  .CLK     ( CLOCK ),
  .RST_X,
  .CNT_EN,
  .CNTOUT
);
```

ポート名とネット名が異なる場合は，名まえによるポート接続で記述する

ポート名とネット名が同一の場合は，ネット名を省略可能

(b) .nameによるポート接続

```
CNT6 CNT6 (
  .*,
  .CLK     ( CLOCK ),
);
```

ポート名とネット名が同一の場合は，.*のみでよい

ポート名とネット名が異なる場合は，名まえによるポート接続で記述する

(c) .*によるポート接続

リスト9　インターフェースの記述

```
interface AMBA_BUS (input wire HCLK, HRESETn);
  wire [31:0] HADDR, HWDATA, RDDATA;
  wire [1:0]  HTRANS, HRESP;
  wire        HWRITE, HREADY, HSELx;
  wire [2:0]  HSIZE, HBURST;
  wire [3:0]  HPROT;

  modport master ( input  HSELx, HRDATA, HREADY, HRESP;
                   output HADDR, HTRANS, HWRITE;
                   output HSIZE, HBURST, HPROT, HWDATA);

  modport slave  ( input  HSIZE, HBURST, HPROT, HWDATA;
                   input  HADDR, HTRANS, HWRITE;
                   output HSELx, HRDATA, HREADY, HRESP);
endinterface
```

インターフェースの入力

インターフェースで束ねる信号

インターフェースに接続するモジュール側で使用する入出力情報

記述します．ポート名とネット名が異なる場合は，.nameと同じように名まえによるポート接続と組み合わせて記述することができます．**リスト8(c)**に，.*を使ったモジュール接続の例を示します．

1.5　インターフェース

図3に示すように，モジュール間の共通信号(バス)はモジュール宣言のたびにすべての信号を記述しなければなりません．モジュールの数が多くなるほど記述は冗長で煩雑なものとなり，バス信号の追加や削除を行う際には非常にたいへんな作業となります．

SystemVerilogでは，新たにインターフェースという概念が取り入れられました．モジュール間で共通の信号をインターフェースで記述して呼び出すことで，記述が簡素化され，修正も楽になります．

● インターフェースの記述

インターフェースの記述を**リスト9**に示します．インターフェースはinterfaceのキーワードで開始し，endinterfaceのキーワードで終わります．

interfaceの後ろには任意のインターフェース名を定義します．かっこ内にはインターフェースの入力信号となるクロックと非同期リセットを定義しています．ここで定義した信号もインターフェースの一部の信号となります．

インターフェースの内部には，インターフェースで束ねる信号の宣言を行います．これはデータ・タイプとビット幅の定義です．その後ろに，インターフェースで束ねた信号の方向(入出力)をmodportのキーワードで定義します．これはインターフェースを呼び出した側がそれぞれの信号を入力として使うのか，出力として使うのかを宣言したものになります．

● インターフェースの呼び出し

モジュールのポート宣言で，インターフェースを使用します．このとき，modportの指定が可能です(**リスト10(a)**)．

インターフェースとモジュールの接続はインターフェー

リスト10　インターフェースの呼び出し

```
interface AMBA_BUS (input wire HCLK, HRESETn);
  modport master ( ... );
  modport slave  ( ... );
endinterface

module IP1 (AMBA_BUS.master bus_con);
  ...
endmodule

module IP2 (AMBA_BUS.slave   bus_con);
  ...
  ...
  ...
endmodule

module CHIP (input wire HCLK, HRESETn);
  ...
  AMBA_BUS BUS (HCLK, HRESETn);
  IP1      IP1 (BUS);
  IP2      IP2 (BUS);
  ...
endmodule
```

ポート名

インターフェースを使ったポートを宣言(modportを指定)

インターフェースのインスタンシエーション

インターフェースのインスタンス名で接続

(**a**) モジュールのポート宣言でmodportを指定した記述

```
interface AMBA_BUS (input wire HCLK, HRESETn);
  modport master ( ... );
  modport slave  ( ... );
endinterface

module IP1 (AMBA_BUS bus_con);
  ...
endmodule

module IP2 (AMBA_BUS  bus_con);
 ...
 always @(posedge bus_con.HCLK or negedge bus_con.HRESETn)
 ...
endmodule

module CHIP (input wire HCLK, HRESETn);
  ...
  AMBA_BUS BUS (HCLK, HRESETn);
  IP1      IP1 (BUS.master);
  IP2      IP2 (BUS.slave);
  ...
endmodule
```

インターフェース内の信号の参照

インターフェースのインスタンス名で接続(modportを指定)

(**b**) モジュールのポート宣言でmodportを指定しない記述

表2　SystemVerilogで拡張された演算子

種　別	演算子	説　明
インクリメント/デクリメント演算子(単項演算子)	++	加算処理
	--	減算処理
代入演算子	+=	右辺を左辺に加算して代入する
	-=	右辺を左辺から減算して代入する
	*=	左辺に右辺を乗算して代入する
	/=	左辺を右辺で除算して代入する
	%=	左辺を右辺で除算して，余りを代入する
	&=	右辺と左辺でビット単位のANDをとり，代入する
	|=	右辺と左辺でビット単位のORをとり，代入する
	^=	右辺と左辺でビット単位のEXORをとり，代入する
	<<=	右辺で示した回数だけ左辺をビット単位で左シフトして代入する
	>>=	右辺で示した回数だけ左辺をビット単位で右シフトして代入する
	<<<=	右辺で示した回数だけ左辺を算術左シフトして代入する
	>>>=	右辺で示した回数だけ左辺を算術右シフトして代入する
比較演算子	=?=	x，zをワイルド・カードとして扱う．一致の場合，真となる
	!?=	x，zをワイルド・カードとして扱う．不一致の場合，真となる

表3　比較演算子の演算結果

A[3:0]	B[3:0]	A=?=B	A!?=B
1010	1010	真	偽
1010	1011	偽	真
10x0	1010	真	偽
10x0	10x0	真	偽
10x0	10z0	真	偽
10z0	10z0	真	偽
10z0	1010	真	偽

スとモジュールを呼び出し，インターフェースのインスタンス名を使って接続します．

モジュールのポート宣言でインターフェースのmodportを指定しなかった場合(**リスト10(b)**)は，インターフェースとモジュールの接続の際に，インターフェースのインスタンス名の後ろにmodportを指定します．

● インターフェース内の信号アクセス

モジュール内でインターフェースの信号を参照または代入する際には，**リスト10(b)**に示すように，ポート名とインターフェースの内部信号名を使った階層パス名でアクセスすることができます．

2 テストベンチ記述

次に，テストベンチ記述向けのSystemVerilog文法について解説します．

2.1　演算子の拡張

表2に示す演算子がSystemVerilogで追加されました．インクリメント/デクリメント演算子および代入演算子は，ブロッキング代入文として動作します．

```
task READ_TASK;
  input   [31:0] ADDR_VAL;
  output         RD_REQ;
  output  [31:0] ADDR;
begin
  RD_REQ  = 1'b1;
  ADDR    = ADDR_VAL;
  @(posedge CLK);
  RD_REQ  = 1'b0;
  ADDR    = 32'h0;
end
endtask
```

値渡し

出力に反映されない

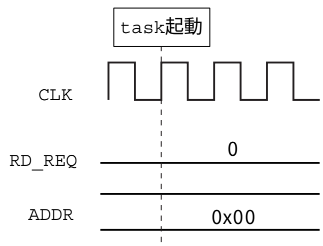
最後の値が出力される

```
task READ_TASK;
  input   [31:0] ADDR_VAL;
  ref            RD_REQ;
  ref     [31:0] ADDR;
begin
  RD_REQ  = 1'b1;
  ADDR    = ADDR_VAL;
  @(posedge CLK);
  RD_REQ  = 1'b0;
  ADDR    = 32'h0;
end
endtask
```

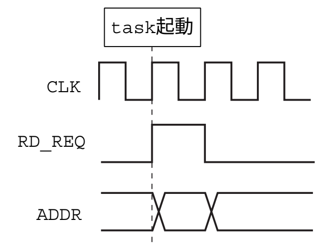
値渡し

参照渡し

出力に反映される



（a）値渡しの記述と出力波形



（b）参照渡しの記述と出力波形

図4
値渡しと参照渡し
(a)のoutput宣言された信号は値渡しになるため，タスク起動中に代入された値は反映されない．(b)のref宣言された信号は参照渡しとなり，タスク起動中でも代入した値が反映される．

また，比較演算子（=?=, !?=）は，if文を使ってxやzをドント・ケア（don't care）として比較することができます．演算結果については，**表3**を参照してください．

2.2　ランダム関数の拡張

Verilog HDLには32ビット符号付きランダム値を生成するシステム関数$random()がありました．SystemVerilogではさらに，制約条件を含んだランダム関数の拡張が行われています．

現時点でツール側の対応が遅れていることもあり，ここでは以下の2種類のランダム関数についてのみ触れます．

- $urandom
- $urandom_range()

記述例については，ツールによる動作確認を行っていませんので，ご注意ください．

● $urandomと$urandom_range( )

$urandomと$urandom_range()は，ともに32ビット符号なしのランダム値を生成するシステム関数です．$urandomはかっこ内にシードの設定が可能です．シードを設定することにより，毎回同じランダム値を生成することができます．シードを省略した場合，ツールに依存した値となります．

$urandom_range()では，ランダム値の取りうる範囲を指定することができます．例えば**リスト11**のように，

リスト11　$urandomと$urandom_range( )の記述

```
logic [ 2:0] ADDR, MODE;
logic [31:0] WRDATA;

initial begin
  $urandom(100);
  ...
  for (...) begin
    ...
    REG_WRITE_RAND;
    ...
  end
end

task REG_WRITE_RAND;
begin
  #(P_DELAY);
    CS_X   = 1'b0;
    WR_X   = 1'b0;
    ADDR   = $urandom;
    MODE   = $urandom_range(4,0);
    WRDATA = $urandom;
  @(posedge CLK) #(P_DELAY);
    CS_X   = 1'b1;
    WR_X   = 1'b1;
end
endtask
```

シードを100としてランダム値を生成する

MODEは0～4の範囲でランダム値を生成する

MODE信号について0～4の範囲でランダム値を生成したい場合，$urandom_range(4,0)と記述します．

2.3　タスクとファンクションの拡張

タスクとファンクションについて，SystemVerilogでは以下のような拡張が行われました．

● 引き数の参照渡し（ref）

タスク，ファンクションの引き数についてinput，

3

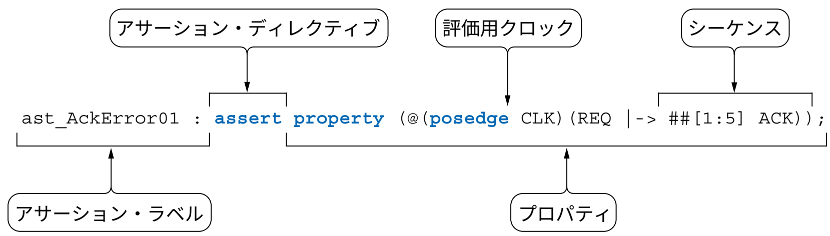


図5
アサーションの構成要素
アサーションの記述は，アサーション・ラベル，アサーション・ディレクティブ，プロパティ，シーケンスで構成される．

output，inoutの宣言が可能ですが，これらの引き数は値渡しになります．すなわち，入力となる引き数は，タスクやファンクションが呼び出された時点で値のコピーがタスクやファンクションに与えられ，出力の引き数はタスクやファンクションが終了した時点の値が引き渡されます．したがって，タスクやファンクションを実行中にリアルタイムに値の入出力を行うことはできません．

SystemVerilogでは，新たにrefというキーワードが追加されました．このrefで宣言された信号は，タスクやファンクション実行中でも入力の変化や出力の変化を反映することができます．これを参照渡しと言います．

図4は値渡しと参照渡しを比較した例です．図4(a)の出力はタスク終了時の値が出力されるため'0'固定となりますが，図4(b)の出力は，タスク処理中にも値が変化します．

● 引き数の名まえ渡し

Verilog HDLでは，タスクやファンクションの呼び出しで引き数を与える場合，仮引き数の定義順に引き数を記述していました．このような記述は誤りやすく，また引き数の順番を誤った場合は発見しにくいバグとなってしまいます．

SystemVerilogでは仮引き数の定義順ではなく，以下のように仮引き数の名まえを使用することができ，記述の誤りを減らすことができます．

以下の記述は図4のタスクを引き数の名まえ渡しで呼び出した例です．

```
READ_TASK(.ADDR_VAL(VALUE),
          .RD_REQ(RD_REQ), .ADDR(ADDR));
```

3 アサーション記述

アサーションには，イベント・ベースで動作する即時アサーションと，サイクル・ベースで動作する並列（コンカレント）アサーションの2種類があります．即時アサーションはレーシングの問題で誤動作することがあり，使用するうえで注意が必要です．一方，並列アサーションはサンプリング用のクロックを持ち，安定した動作が得られます．

ここでは並列アサーションについて説明します．即時アサーションについては，「文法ガイド編」を参照してください．

3.1 アサーションの構文

アサーションは，「プロパティ（回路仕様）に違反した動作をしていないか？」，または「プロパティが実行されたか？」をチェックする機能を持ちます．プロパティに違反した動作を検出した場合はメッセージを出力し，回路動作で問題が生じたことを設計者に通知します．

まず，図5の記述例を使ってアサーションの構文を見てみることにします．アサーションは，以下の構成要素で記述されます．

- アサーション・ラベル
- アサーション・ディレクティブ
- プロパティ
- シーケンス

● アサーション・ラベル

アサーションの先頭のアサーション・ラベルは，設計者が任意に記述することができます．アサーションで出力されるメッセージには，このアサーション・ラベルが付加されます．アサーションごとにわかりやすいラベル名を付けておいたほうがよいでしょう．

● アサーション・ディレクティブ

アサーション・ディレクティブは，プロパティの真偽によってアサーションの動作を決めるもので，以下の3種類があります．

- assert：プロパティが偽の場合，エラーと判断する

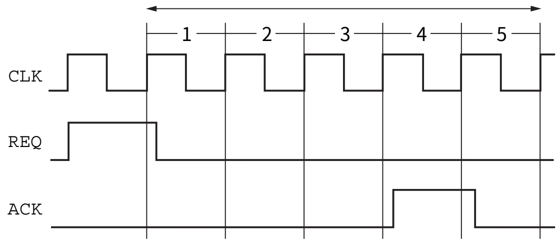


図6　REQに対するACKのレスポンス時間
図5のアサーションの記述は，上記波形に示すREQとACKの関係をチェックしている．ACKが5サイクル以内に'1'になれば，プロパティは真となる．そうでない場合はプロパティは偽となり，アサーションはエラー処理を行う．

- assume　：プロパティが偽の場合，エラーと判断する
- cover　　：プロパティが真となった回数を検出する

assertとassumeは同じ動作になっていますが，assumeは回路入力の値の範囲やタイミングを定義するもので，おもに形式的検証（フォーマル・ベリフィケーション）で使用されます．入力パターンを使ったシミュレーション動作はassertもassumeも同じになります．

coverは，assertやassumeとは異なり，シミュレーション時にプロパティが実行されたことを検出し，シミュレーション終了時にその回数をレポートします．もしプロパティが1回も真になっていない場合は，シミュレーション・パターンが不足していることがわかります．すなわちシミュレーション・パターンの過不足を判断する機能カバレッジとして使用されます．

● プロパティ記述とシーケンス記述

プロパティ記述は，回路仕様を明記したものです．まず，プロパティには評価用のクロックを記述します．**図5**の記述例では，CLKの立ち上がりでプロパティが評価されます．

プロパティについては，ビット演算子や論理演算子を用いて，単純なブール式で表現することもできますし，プロパティ演算子やシーケンス演算子を用いて，複数サイクルにまたがった条件式の記述を行うこともできます．

シーケンスはプロパティを構成する要素で，シーケンス演算子を用いて時間の概念を持った条件式を記述することができます．

プロパティ演算子とシーケンス演算子については，「文法ガイド編」を参照してください．

リスト12　状態遷移をチェックするアサーション

```
cvr_Cond_NORMAL_to_SEC :
cover property (@(posedge CLK)
        $past(SW2, 1) and
        ($past(CUR_STATE, 1) == NORMAL_ST) and
        (CUR_STATE == SEC_ST));
```

1サイクル前にSW2=1である（$past(SW2, 1)）
1サイクル前の状態がNORMAL_STである（$past(CUR_STATE, 1) == NORMAL_ST）
現在の状態がSEC_STである（CUR_STATE == SEC_ST）

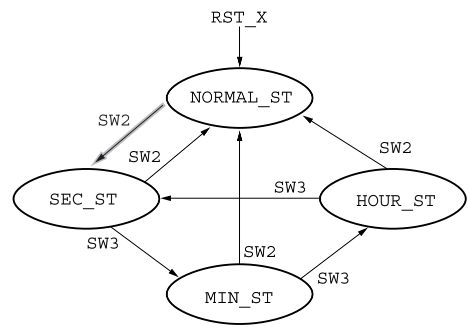


図7　状態遷移図
NORMAL_STからSEC_STは，SW2が'1'になることで遷移する．リスト12はこの状態遷移をチェックするアサーションの記述である．

3.2　assertを使ったアサーションの記述例

では，先ほどの**図5**で示した記述の内容について説明します．このアサーションはassertを使用しているため，クロックの立ち上がりでプロパティが偽となった場合に，アサーションはエラーとして動作します．

プロパティでは，(REQ |-> ##[1:5] ACK)という記述がありますが，この記述は，**図6**で示す波形のようにREQが'1'になってから5サイクル以内にACKが'1'になることを意味しています．

|->は，この記号の左の条件が成立した同じサイクルで右の条件を評価するプロパティ演算子です．##[1:5]は，1～5サイクルの遅延を意味するシーケンス演算子です．まとめると，このアサーションは，REQが'1'になってから5サイクル以内にACKが'1'になる必要があることを意味しています．

5サイクル以内にACKが'1'にならない場合，プロパティが偽となり，アサーションはエラーが発生したことを設計者に通知します．

3.3　coverを使ったアサーションの記述例

リスト12に示す記述は，**図7**に示すステート・マシンの

状態遷移(NORMAL_ST → SEC_ST)が実行されたことをチェックするアサーションです．アサーション・ディレクティブとしてcoverを使っているため，このアサーションはクロックの立ち上がりでプロパティが真になった回数をチェックし，シミュレーション終了後に設計者に結果を通知します．

プロパティの記述の中でシステム関数$pastが使われていますが，$past(SW2, 1)は1サイクル前のSW2の値を返します．よって，プロパティの記述は，「現在の状態がSEC_STであり」かつ「1サイクル前の状態がNORMAL_STであり」かつ「1サイクル前のSW2の値が'1'である」という条件になります．

シミュレーション終了時に，もし1度もこのアサーションのプロパティが真にならなかった場合，シミュレーションでNORMAL_STからSEC_STへの状態遷移が生じなかったことがわかります．

このようなアサーションは，シミュレーション・パターンが十分であるかどうかを判断する際に非常に役に立ちます．

参考・引用*文献

(1) Stuart Sutherland, Simon Davidmann, Peter Flake(浜口加寿美，河原林政道，高嶺美夫，明石貴昭 訳)，SystemVerilogによるLSI設計，丸善，2005年．

(2) Harry D. Foster, Adam C. Krolnik, David J. Lacey(東野輝夫，岡野浩三，中田明夫 監訳)，アサーションベース設計，丸善，2004年．

こんどう・ひろし
(株)エッチ・ディー・ラボ
kondo@hdlab.co.jp

<筆者プロフィール>
1991年，日本電装入社．1997年に退社した後，エッチ・ディー・ラボに入社．ディジタル携帯電話，無線LAN，Bluetoothなどの無線機器用LSI(制御・変復調)やキャッシュ・メモリの開発に携わる．現在，SystemVerilogのトレーニング・コースを立ち上げ中．

文法ガイド編

1 SystemVerilogで拡張された文法

SystemVerilogで拡張された文法について，要点と構文をまとめます．省略可能な項目については，[ ]を付けています．

1.1 データ・タイプ

● 基本データ・タイプ

表1(p.58)を参照．

レジスタ型，ネット型として使用可能．解決機能付きタイプではないため，複数のドライバを持つことはできない．

● unsigned

```
<データ・タイプ> unsigned <信号名>;
```

符号付きデータ・タイプを符号なしデータ・タイプとして宣言する修飾子．

● signed

```
<データ・タイプ> signed <信号名>;
```

符号なしデータ・タイプを符号付きデータ・タイプとして宣言する修飾子．Verilog HDL 2001で拡張された．

● ユーザ定義タイプ

```
typedef <データ・タイプ> <タイプ識別子>;
```

SystemVerilogのデータ・タイプを使ってユーザが任意のデータ・タイプを作成することができる．

● 列挙型

```
enum [データ・タイプ] { <列挙名> [=定数],
                        <列挙名> [=定数], … };
```

取りうる値の集合を列挙名のリストとして宣言する．列挙名のデータ・タイプは，デフォルトでint型．enumの後にデータ・タイプを宣言することで，列挙名のデータ・タイプの変更が可能．

列挙名の値は，デフォルトで0から順番に割り振られる．ユーザが列挙名の後に定数値を記述することで，列挙名の値を変更できる．

1.2 配列，構造体，共用体

● アンパック型配列

```
<データ・タイプ> <要素のビット数> <配列名> <要素の数>;
```

配列の要素は独立して存在する．全データ・タイプに対してアンパック型配列の宣言が可能．

● パック型配列

```
<データ・タイプ> <要素の数> <要素のビット数> <配列名>
```

配列の要素は，個々の配列の要素が連続したベクタとして存在．ビット単位のデータ・タイプ(logic, bit, reg, wire)，およびパック型配列，パック型構造体，パック型共用体に対して宣言が可能．